EAST WEST UNIVERSITY

Department of CSE

Course Code: CSE246
Course Title: Algorithms
Project Report

Project Title: Matrix Chain Multiplication

Submitted by-

Name	Id
Md. Naimul Islam	2020-2-60-173
Sumaya Akter	2019-1-67-056
Noushin Pervez	2020-1-60-189

Section: 3
Summer 2022

Submitted to-
Jesan Ahammed Ovi
Senior Lecturer
Department of Computer Science and Engineering
East West University

Submission Date: 11-09-2022

Matrix Chain Multiplication

Problem Statement:

Given a sequence of matrices, we have to find the most efficient way to multiply these matrices together. However, the problem is not actually to perform the multiplications, rather we have to decide in which order the multiplications should be performed. Since the multiplication is associative, there are so many options to multiply these matrices. In other words, no matter how we parenthesize the product, the result will be the same. However, the order in which we parenthesize the product affects the number of simple arithmetic operations needed to compute the product, or the efficiency.

We have to use dynamic programming (DP) approach to find the best association such that the result is obtained with the minimum number of arithmetic operations.

System Requirements:

We used the Ryzen 5 5500U processor with 8GB RAM in Windows 10 Home. We used Codeblocks IDE. We can also run this program on any lower or higher end devices.

System Design:

Chain means one matrix's column is always equal to the second matrix's row. If A is a $(p \times q)$ matrix, B is a $(q \times r)$ matrix, C is a $(r \times s)$ matrix. The cost of multiplying A, B, C is-

$$(AB)C = pqr + prs$$
$$A(BC) = qrs + pqs$$

When $p = 4, q = 5, r = 6, s = 2$

$$(AB)C = 120 + 48 = 168$$
$$A(BC) = 60 + 40 = 100$$

We can see from the example that the multiplication sequence or parenthesization is important. An optimal order minimizes the total number of operations. Dynamic Programming solution involves breaking up the problems into subproblems whose solution can be combined to solve the global problem to minimize the cost of multiplication. To get the optimal sequence of multiplications we need-

- To split the chain and
- To parenthesize the sub chains

We know that, for some k,

$$dp[i, j] = \min(dp[i, k] + dp[k + 1, j] + arr[i - 1] * arr[k] * arr[j]) \text{ if } i < j$$

To calculate dp[i, j], we already calculated dp[i, k] and dp[k + 1, j]. When a sequence is calculated using recurrence relation, all the values needed by the recurrence relation have already been calculated. In this way, each subproblem is used to fill the table without recomputing it again and again.

Implementation:

```
struct dimension{
    int a;
    int b;
};
```

We declared a structure named dimension to store the values of rows and columns of matrices.

int main() function-

```
struct dimension array[n];
for(int i=0 ; i<n ; i++){
        cin>>array[i].a;
        arr[i] = array[i].a;
        cin>>array[i].b;
}
arr[n] = array[n-1].b;
```

In the main function, we initialized a dimension type array and stored all the information of matrices. arr[n + 1] stores the number of rows of all matrices and the column of the last matrix.

void multiplication(int n, int arr[]) function- (for minimum number of arithmetic operations)
In the parameter, we get (n + 1) and the arr[].

```
for(int d = 1 ; d < n-1 ; d++){
        for(int i = 1 ; i < n-d ; i++){
            j = i+d;
            min = INT_MAX;
            for(int k=i ; k <= j-1 ; k++){
                q = dp[i][k] + dp[k+1][j] + (arr[i-1]*arr[k]*arr[j]);
                if(q < min){
                    min = q;
                    traceArr[i][j] = k;
                }
            }
            dp[i][j] = min;
        }
}
cout<<dp[1][n-1];
```

We start filling the table away from the diagonal. The table will be an upper triangular matrix.

1. The outermost for loop is to iterate diagonally. The value, d iterates from length 1 to length (n - 2).
2. The middle for loop is for the rows. This value, i iterates from 1 to (n - d - 1).
3. The innermost for loop is for columns. The value, k iterates from i to (j - 1).

Diagonal indices can be represented as-

- When d = 1 and i = 1, j = i + d = 1 + 1 = 2 which becomes dp[1][2]
- When d = 1 and i = 2, j = i + d = 2 + 1 = 3 which becomes dp[2][3]
- When d = 1 and i = 3, j = i + d = 3 + 1 = 4 which becomes dp[3][4]

  .

  .

  .
- When d = n - 2 and i = 1, j = i + d = 1 + n - 2 = n - 1 which becomes dp[1][n - 1]

At first, we initialize the array to maximum integer value (INT_MAX) so that the minimum cost will definitely be the calculated one.

We can see that k will have only one value, if j = 2. The loop will only iterate for only one time and that will be the result. If the value of j = 3, then the loop will iterate for two times and the minimum cost will be compared and memorized. Thus, the loop will iterate for k times for each value of (j - 1).

The optimal solution must break at some point, k. The best way to multiply the chain from i to k and from (k + 1) to j and the cost of the final product, we use the formula for dynamic programming. We are getting the minimum value by comparing the previously calculated value and the newly calculated value.

We are also memorizing the value of k for which we got the minimum cost. It stores the optimal break point for every subexpression (i, j) in traceArr[i][j]. It determines the optimal splitting. The array can be used to recover the multiplication sequence. Finally, the value will be minimum when the loop ends. It prints the value of minimum operations which is stored in dp[1][n - 1].

void trace(int i, int j) function- (for parenthesization)

```
if(i == j){
	cout<<x;
	x = (int)x + 1;
	return;
}
```

```
else{
        cout<<"(";
        trace(i, traceArr[i][j]);
        trace(traceArr[i][j]+1, j);
        cout<<")";
}
```

We traverse the traceArr[n][n] array and print parenthesization in subexpression (i, j). In the parameters, we get the initial value of the first matrix (1) and the last matrix (n).

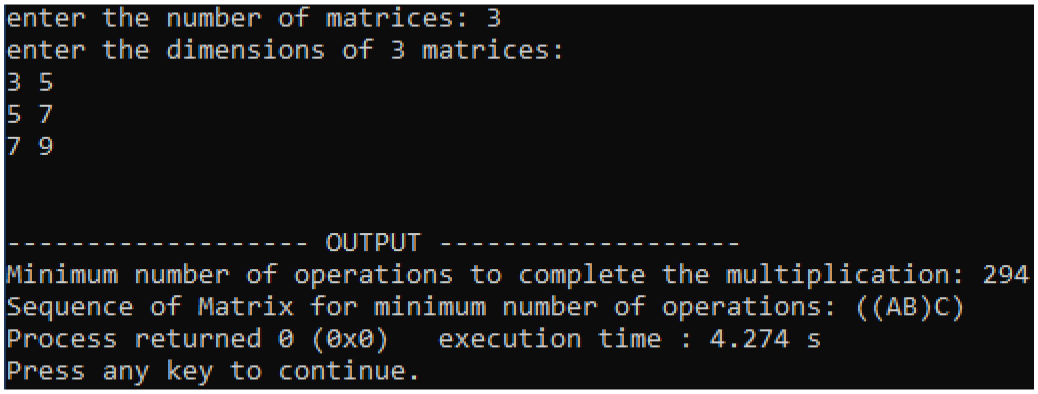
There are two steps here. If i = j (on diagonal 0 or on traceArr[i][i]), we print the matrix. Otherwise, we execute the four steps. It has two recursive calls. The first call recursively puts brackets around subexpression from i to traceArr[i][j]. The second call recursively puts brackets around subexpression from (traceArr[i][j] + 1) to j. This function prints necessary brackets and matrices creating a tree.

Time Complexity:

There are three nested loops. Each loop iterates for ≤ n values. Therefore, the time complexity is O(n^3).

Testing Results:

Here, we take 3 matrices as input. A is a 3 ✕ 5 matrix, B is a 5 ✕ 7 matrix and C is a 7 ✕ 9 matrix. It prints the minimum number of arithmetic operations and the sequence of matrices for the resulting minimum number of operations. The minimum number of operations is 294 and the sequence of parentheses of the product is ((AB)C).

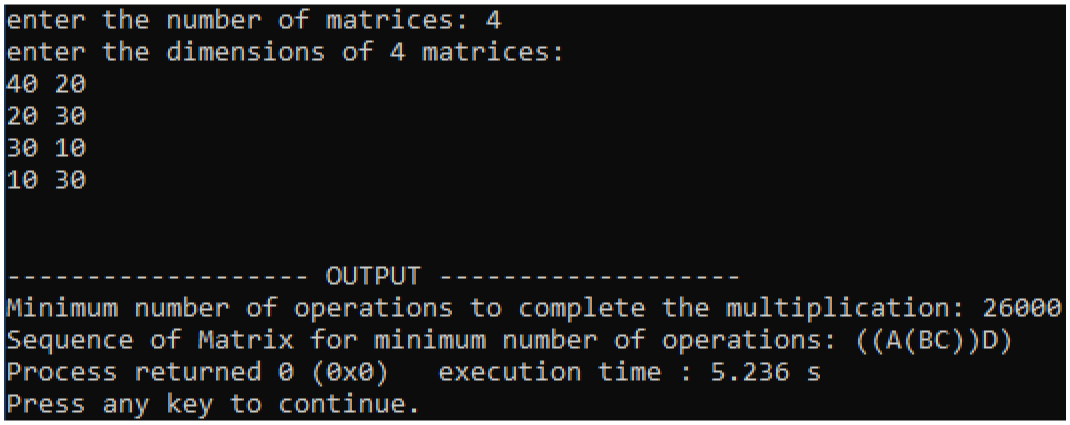


```
enter the number of matrices: 3
enter the dimensions of 3 matrices:
3 5
5 7
7 9



------------------- OUTPUT -------------------
Minimum number of operations to complete the multiplication: 294
Sequence of Matrix for minimum number of operations: ((AB)C)
Process returned 0 (0x0)   execution time : 4.274 s
Press any key to continue.
```

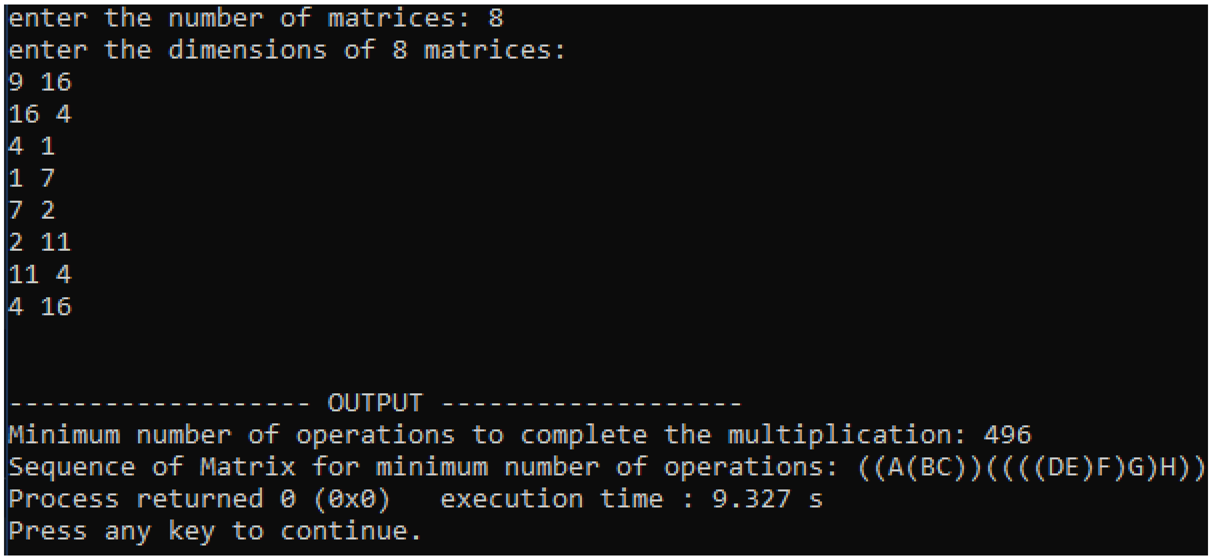
We take 4 matrices as input. A is a 40 ✕ 20 matrix, B is a 20 ✕ 30 matrix, C is a 30 ✕ 10 matrix and D is a 10 ✕ 30 matrix. The minimum number of operations is 26000 and the sequence of parentheses of the product is ((A(BC))D).



```
enter the number of matrices: 4
enter the dimensions of 4 matrices:
40 20
20 30
30 10
10 30


------------------- OUTPUT -------------------
Minimum number of operations to complete the multiplication: 26000
Sequence of Matrix for minimum number of operations: ((A(BC))D)
Process returned 0 (0x0)   execution time : 5.236 s
Press any key to continue.
```

Again, we take 8 matrices as input. A is a 9 ✕ 16 matrix, B is a 16 ✕ 4 matrix, C is a 4 ✕ 1 matrix, D is a 1 ✕ 7 matrix, E is a 7 ✕ 2 matrix, F is a 2 ✕ 11 matrix, G is a 11 ✕ 4 matrix, H is a 4 ✕ 16 matrix. For this test case, the minimum number of operations is 496 and the sequence of parentheses of the product is ((A(BC))((((DE)F)G)H)).



```
enter the number of matrices: 8
enter the dimensions of 8 matrices:
9 16
16 4
4 1
1 7
7 2
2 11
11 4
4 16


------------------- OUTPUT -------------------
Minimum number of operations to complete the multiplication: 496
Sequence of Matrix for minimum number of operations: ((A(BC))((((DE)F)G)H))
Process returned 0 (0x0)   execution time : 9.327 s
Press any key to continue.
```

Future Scope:

We calculate the minimum number of arithmetic operations in this Matrix Chain Multiplication problem. We follow dynamic programming and recursion to solve the matrix chain multiplication problem. The matrix chain multiplication can be generalized to solve more abstract problems. Given a linear sequence of objects, we perform an associative binary operation on those objects as well as all partial results. To apply the operation over the sequence, we compute the minimum costing way to group the objects.